Use Cases -- Yesterday, Today, and Tomorrow

by **Ivar Jacobson**
Vice President
Process Strategy
Rational Software
IBM Software Group

To my knowledge, no other software engineering language construct as significant as use cases has been adopted so quickly and so widely among practitioners. I believe this is because use cases play a role in so many different aspects of software engineering.

Although I first used the term in 1986, I had actually been working on and evolving the concept of use cases since 1967. So many people have asked me how I came up with this concept that I decided to write this article to explain the origins and evolution of use cases. I'll also summarize what they have helped us achieve so far, and then suggest a few improvements for the future.

Yesterday: In the Beginning

It was 1986; I'd worked at Ericsson for more than twenty years and was trying to figure out a better way to model telephone calls. The modern switches of that time handled so many different types of telephone calls: local calls; outgoing calls; incoming calls; transit calls. There were also many kinds of local calls and many kinds of outgoing calls: calls to a neighbor switch; calls to a domestic switch; calls to an international switch. And, on top of this, each one of these calls could be carried out with different signaling systems.

We had been grappling with the problem of modeling multiplicity and diversity for many years. It would have been very difficult to model each

type of call -- there were too many, and there was a lot of overlap between them. So first we listed and named them. We called them "traffic cases." *Our approach was to model the different "functions" or "features" needed to carry out all the calls -- all the traffic cases*. Functions had no interfaces. They had beginnings and endings, but they were not well defined. A function could interact with the outside world. The general feeling was that we didn't really know what functions were, but we could give examples of them, and some people could specify them.

However, we did know how to realize functions. I had learned a diagramming technique to describe sequences of relay operations; in 1969 I translated this technique to describe software component interactions. I called the resulting diagrams "sequence diagrams," and we still use that name for them today. With sequence diagrams (or collaboration diagrams for simpler interactions) we described how functions were realized, in very much the same way we describe use-case realizations today.

Then, one day in spring of 1986, while working on traffic cases and trying to map them onto functions, I suddenly "got it." I could describe a traffic case in terms of functions by using an inheritance-like mechanism. I began calling both *traffic cases* and *functions* "use cases"-- the former became *concrete* or *real* use cases, and the latter became abstract use cases.

I introduced these new constructs in a paper for OOPSLA'86, but it was not accepted (probably because I'd already submitted another paper that was accepted, or because most people on the program committee were programming language experts). However, an updated version introducing many key ideas for use-case modeling was accepted the next year, for OOPSLA'87.

What Was a Use Case in 1987?

According to the OOPSLA'87 paper, "*A use case is a special sequence of transactions, performed by a user and a system in a dialogue*." This is pretty similar to our current (informal) definition. I developed a separate model for describing a system from an *outside* perspective and called it a *use-case model*. This provided a *black-box view* of the system -- the system's internal structure would be of no interest in this model. Some people have misunderstood the term *outside*, mistaking it for a synonym for user interface. Instead, the use-case model represents the functional requirements of the system.

At this time the use-case model also included entity (domain) objects, so we could show how use cases could <<access>> entities. Use cases and entities were class-like; they had operations and data. The other relationship depicted in the use-case model was <<built-on>> which was described as "*...an extended form of inheritance relation. Multiple inheritances are common*." In fact, the built-on relationship was a combination of the generalization relationship and the <<extend>> relationship.

After the use cases were specified, they were also designed and tested.

> You create as many processes [today we would say *activities*] as there are use cases. The conceptual model of the use cases is translated seamlessly into a new model showing how each use case is implemented by means of the identified blocks [today a *block* would be a subsystem, class, or component]...Each use case is tested separately to safeguard that the system meets the requirements of the user. Please, note that the use cases constitute the key aspect through the entire [set of] development activities.

Sequence diagrams were used to show interactions among the blocks/components. This was no surprise, since sequence diagrams had shown their value in practice for almost twenty years prior to that time.

What Was a Use Case by 1992?

As you can see, use cases had assumed much of their current shape (syntax and semantics) by this time. Between 1987 and 1992, they evolved and matured, as about twenty customers put the Objectory Process[1] to practical use for new product development. These customers were involved in many different kinds of systems: management information; defense (pilot, counter measure, C3I); and telecom (POTS, mobile). Use cases took on a new relationship: "inheritance" (now replaced by "generalization"). I avoided what is now called the <<include>> dependency, which I thought would damage modeling by inviting functional decomposition.

To increase clarity, we made it an important issue to distinguish between a *use case* (as a class-like thing), an *instance* of a use case, and a *description* of a use case.

The depth of the use-case model was in its use-case descriptions. Each use-case description contained the following:

- A brief description of the purpose of the use case
- A flow of control
- Base flows and alternative flows
- Subflows (reusable in many places within the same use-case description)
- Preconditions and postconditions

The use case was more than just a requirements technique; it was like the hub of a wheel:[2]

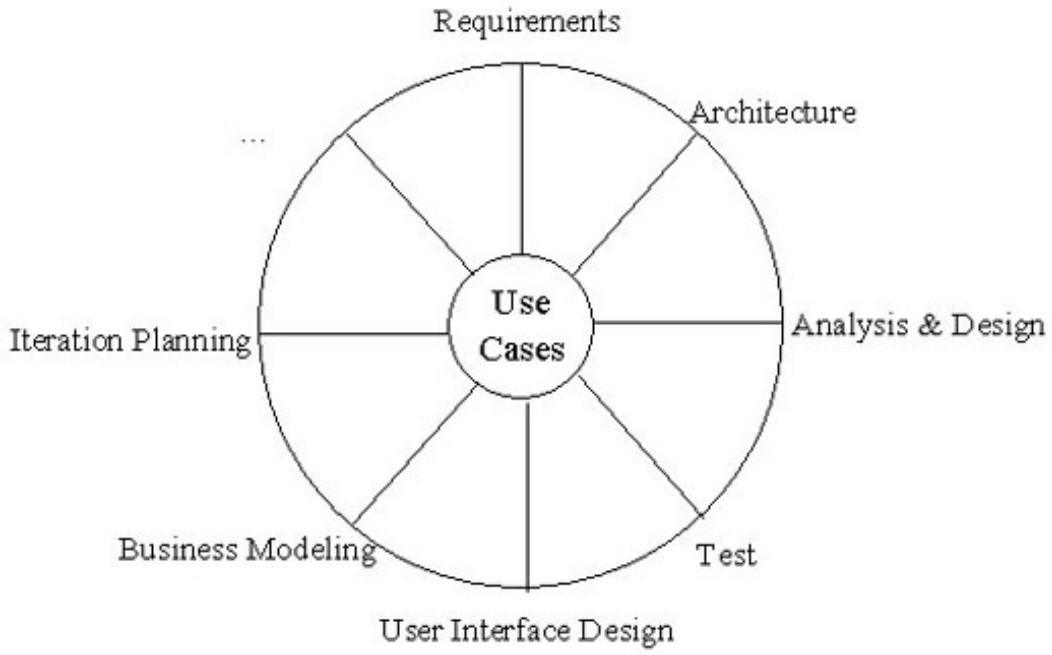

Figure 1: Use Cases Were Like the Hub of a Wheel

Use cases were traceable to analysis, design, implementation, and test. For each use case in the model, we created a collaboration (a view of participating classes) in analysis and design. Each use case resulted in a set of test cases. Use cases were important in designing user interfaces and structuring the user manual. Use cases also moved into the space of business modeling, since they could perfectly capture business processes.

We coined the term *use-case driven development* for our approach. First, you identified all use cases and specified each one of them in *requirements*. You analyzed and designed each use case in analysis and design respectively, and finally tested each and every one in *test*.

And we had all this before 1992! In that year, together with my colleagues, I authored a book called *Object-Oriented Software Engineering: A Use Case Driven Approach*.[3] What was presented at OOPSLAý87 was theory; based on our work with customers,[4] we now had a lot of practical experience to back up that theory.

Today: A Lot Has Happened Since Then

The adoption rate of use cases has surprised me: They were embraced almost immediately by all methodologists and basically adopted worldwide. Other important techniques such as component-based design and object-oriented modeling were much more controversial and needed a much longer adoption time. Probably this is because use cases are basically a simple and obvious idea; they work well with objects and object thinking. Using use cases is not just a technique for managing requirements; it binds together all the activities within a project, whether a mini-project such as a single iteration or a major project resulting in a new product release.

How Many Use Cases Is Enough?

The current definition of use cases basically goes back to 1994. To help teams strike a balance between defining too many use cases or too few, I added a requirement that a use case must give a "*measurable value*" to a "*particular actor.*" As a rule of thumb, I suggested that a large system supporting one business process should have no more than twenty use cases. I realized that giving any such number could lead people to take undesirable actions to get the "right" number. If they had fewer than twenty use cases, they might split some of them to get up to twenty. Or, if they had more than twenty use cases, they might combine use cases to bring down the count. But that is the wrong approach. I have seen good use-case models for commercial systems with as few as five use cases, and some with as many as forty use cases. However, I have also seen use-case models with as many as 700 use cases! Obviously, these were unsound models. **The twenty use cases I suggest should all be concrete (real) use cases and not generalizations or extension/inclusion fragments.**

Use Cases and the Unified Modeling Language

Use cases have become part of the Unified Modeling Language (UML). And because the UML is precisely defined, use cases and associated concepts (such as use-case instance [UCI]) are also now precisely defined, thanks to the UML's powerful classifier concept (e.g., what I called "class-like" in 1992 can now be formally explained by UML classifiers). Although the old definition that "a use case is a sequence of actions..." is still compatible with the user's perspective, the definition based on classifiers is what methodologists, process engineers, and tool builders need for clarity.

Note, however, that although the UML effort resulted in a much more precise definition of use cases, it didn't do much to evolve them. Roughly speaking, we only changed the "uses" relationship to a generalization, and we added an <<include>> relationship. The "uses" relationship in the Objectory Process was previously called "inheritance" and was never intended to be used for <<include>> fragments. In the past, we didn't allow developers to model these fragments, but used another technique involving text objects instead. We'll discuss these text objects later.

Since my company became part of Rational, I have been very happy with the way our Rational Unified Process,® or RUP® team has correctly implemented use cases and improved their practical use. Although they have made no really dramatic changes, they have provided much better explanations of use cases, based on the experience of thousands of customers and our own experts. In particular, a new book, *Use Case Modeling*,[5] by Kurt Bittner and Ian Spence, is now on the shelves. This is *THE* book on use cases. I strongly recommend that everyone involved in software engineering and requirements development read it. Also, work by Jim Conallen[6] and by Peter Eeles, Kelli Houston, and Wojtek Kozaczynski[7] on user experience design with use cases is a great improvement on our earlier work in this area, and is very much in line with the original use-case concept.

Now may be the time to take steps to grow (clarify and extend) the idea of use cases. But first, a word of warning about formalizing use cases.

Use Caution When Formalizing Use Cases

Over the years, people have complained that there was not information in the UML on how to formalize use cases. Although several of my papers discuss techniques for doing this, such as using sequence diagrams to show how an actor interacts with a use case, or using activity diagrams or state charts to describe a single use case, I warned against using these techniques. After all, *the use-case model is intended for communicating with customers and users*. Formalizing use cases (using mathematics) has never been a problem. I had already done it in 1986. In fact, any computer science student could do it. By making use cases classifiers in UML, you have the tool to formally describe use cases in basically any depth you want.

The challenge is to use, in the most pragmatic way, what the UML makes available. *I am still reluctant to suggest that system analysts describe use cases in a more formal way than simple text*. Avoid trying to specify the *internals* of each use case with diagrams such as activity diagrams or state charts, although it is good to describe the *interactions* between a use case and actors with sequence diagrams or activity diagrams with swimlanes. I think there are better ways to become more precise about requirements (the internals of a use case) than introducing more formalism into the use-case model. But that is the role of the analysis discipline -- and the subject of another article.

Tomorrow: Potential Next Steps

Over the last decade, techniques for writing use cases have remained quite stable. Although I have been tempted to make improvements, as soon as people began discussing them, everything suddenly became open to questioning and too unsettled. In the end, I felt it was safer to leave things as is, until people become more familiar with the use-case construct. Also, we needed to allow time for use cases to be used in the field, and for their implementation to evolve and become established.

In this section, however, I will raise a couple of issues regarding the current application of use cases and propose possible changes.

A use-case model of a software system contains basically four kinds of use cases:

- **Concrete** use cases that can be instantiated (abstract ones can't).
- **Generalization** use cases that support reuse of use cases.
- **Extension** use cases that add behavior to an existing (or presumed) use case, without changing the original use case.
- **Inclusion** use cases that add behavior to other use cases by changing them.

We will discuss these different types of use cases below.

Generalizations. These use cases are abstract; they are **generalizations of either concrete use cases (through the generalization relationship) or other abstract use cases.** The generalization use case (parent use case) and its sub-use cases (children) should be of the same type to maintain substitutability -- in other words, you should be able to use an instance of a child whenever you expect an instance of the parent. Strictly speaking, you might not always be able to do this with use cases (or any state-driven classifier), since a child use case can require some extra interaction with the actors. However, making the child the same type (classification) as the parent is still important. For example, Make a Local Call (child) and Make a Wake-Up Call (child) would both be generalized to the abstract use-case Make a Call (parent).

Extensions. Recall that extension use cases serve a very special purpose: They **add behavior to an existing (or presumed existing) use case without changing it.**[8] Using extensions is a technique to get easy-to-understand descriptions. First you describe the basic (mandatory) behavior, and then you add extra (mandatory or optional) behavior--behavior that is not needed to understand more basic behavior. *Extensions are not just a technique for describing optional behavior (optional, that is, from the customer's point of view); they also describe mandatory behavior in a structured way*. Without a mechanism such as extensions, the base flow of a use case would become cluttered with statements that have nothing to do with the base use case, even if the statements are important for other use cases.

A potential problem in using extensions is *creating deep hierarchies of extend dependencies*. To avoid doing this we follow a guideline: We *usually never extend an extension (a fragment)*, since that would make the results difficult to understand.

Another potential point of confusion is knowing when to use extensions and when to use alternative paths when describing a use case. Again, we need guidelines: We usually use the extend relationship only when the *extension use case is completely separate from the extended base use case* -- or, more precisely, when it is a separate, concrete use case in itself, or when it is only a small fragment that is needed by another use case. *The base use case must be complete by itself and not require the extension. Otherwise, you must use alternative paths to describe additional behavior.*

Extensions can help a lot in managing software development over the entire software development lifecycle. For example, you can add a large class of extensions without requesting regression tests for the base. You would need only to test the extensions and their cooperation with the existing base. Let me qualify this. First, extensions as language constructs need to propagate through design and implementation: They need to be added to the design model, the implementation model, the executable code, and so on.[9] Second, only extensions that don't access other use cases' objects (more correctly, extensions that don't modify objects shared with other use-case realizations) would belong to this class. When such

conditions are fulfilled, we could prove that some extensions wouldn't be able to damage the existing software.

I proposed the idea of extensions back in 1978 when I worked at Ericsson and then wrote about them for OOPSLA '86,[10] but developers didn't embrace the idea until 1991. Even Ericsson recognized that the idea had merit, though: They applied for patents to support extensions to C++ and to the operating system -- and also to the computer architecture. Their infrastructure team had suggested these to lower development costs. Further discussion is beyond the scope of this article, but the point is this: *Don't think of extensions as useful for use-case modeling only!*

In a future article, I hope to address how extensions could propagate through activities other than use-case modeling -- including analysis, design, implementation, and test. As noted above, our 1992 book covered this territory.

Inclusions. When use cases were born, I saw the need for two kinds of reuse. First, since I based use cases on object-oriented ideas, I saw great value in *subclassing*, and in 1987 I introduced the "*inheritance*" relationship between use cases. In 1992 we changed this to "*uses*" to make it sound less like "techie" jargon and easier to adopt by analysts. Later, the UML called the relationship "*generalization*." Second, the other reuse need was simply a mechanism for factoring common flows of events (sharing) from use-case descriptions; this was for cases in which we currently use the <<include>> relationship and refer to the shared behavior as an *inclusion*.

I was very reluctant to introduce a relationship like this; I foresaw people misusing use cases by applying them for functional decomposition. In fact, today people do misuse use cases by using them to describe functions as opposed to objects; and then they blame the use-case concept for their problem. To get around this, we introduced another idea. In the Objectory tool, we supported reuse through "text objects,"[11] reusable objects consisting of a piece of text. These text objects could be changed only by the person responsible for the text object, not by someone else using (or reusing) the object.

Text objects could be reused in multiple places -- for example in different text descriptions of use cases. Like Rose/XDE, which has a model element for each class that could be shown (differently if necessary) in multiple diagrams, text objects could be shown in multiple documents. The beauty was that all the text objects were kept in one place, so they were easy to find, change, and manage, and all references were automatically kept in sync. Very powerful!

I think the solution we had was right at that time, when we feared that use cases would be viewed as just another way to do functions. This problem persists today among system analysts, although not among methodologists.

Two Classes of Extension/Inclusion Use Cases

When extension use cases originally were introduced, I had basically only one kind of extension or inclusion in mind: the *small reusable use-case fragment*. I didn't foresee the need to extend or include concrete complete use cases. This is something we learned during the first four years of practical use. Many times we[12] discussed the need for two kinds of extension use cases. However, we didn't want to make use-case modeling more complex. During the UML 1.1 work, we (primarily Jim Rumbaugh, Gunnar Overgaard, and I) touched on this subject, but for the same reason we didn't follow through. Maybe now is the time.

Below we will discuss the two classes of extension/inclusion use cases.

1. Extensions/inclusions that are concrete use cases. These use cases interact with and provide value to actors. As an example,[13] consider a "surveillance" system that reports intruders. The base (concrete use case) monitors the surveillance area, and perhaps even does some other work, such as maintaining a constant building temperature. The extending use case (also a concrete use case) reports unusual events -- security breaches or fires -- to the appropriate authorities (police, fire, building management). This illustrates that the base use case has some significant behavior, as does the extending use case. Both are concrete use cases, and both can be instantiated.

2. Extensions/inclusions that are just fragments of a use case. This is a far larger class of use cases, but each member is usually very small. These use cases are abstract, in that they *cannot be instantiated separately*. They are needed by some other use case, usually a concrete or a generalization use case. For example, in Figure 2, let us assume that the base use case is a bank transaction: *Conduct Transaction*. Every time a transaction fails, the bank wants to register this event to make it available to some *other concrete use case*. In this case, that would be an administrative use case: *Inspect Transaction Failures*.

One obvious traditional solution would be to change the transaction use case and show explicitly in its description that a failure message has been registered. However, this would require changing the base and making it more difficult to understand. The change has nothing to do with the base use case *Conduct Transaction*; it's only there to register some information to the other use case. One such change may not be disturbing, but when you have several of them, it gets to be quite messy. To avoid cluttering the base, we instead use an extension use case to add the change *on top* of the base use case. Then, we would add a *third use case* -- a very small use-case fragment called *Register Failures*. Similarly, we may have small, procedure-call like inclusion use cases that are shared between two or more concrete use cases. Assume that *Validate User* is such an included use-case fragment (shared with some other real use case). In contrast to concrete uses cases, both Register Failures and Validate User are *abstract* use cases.

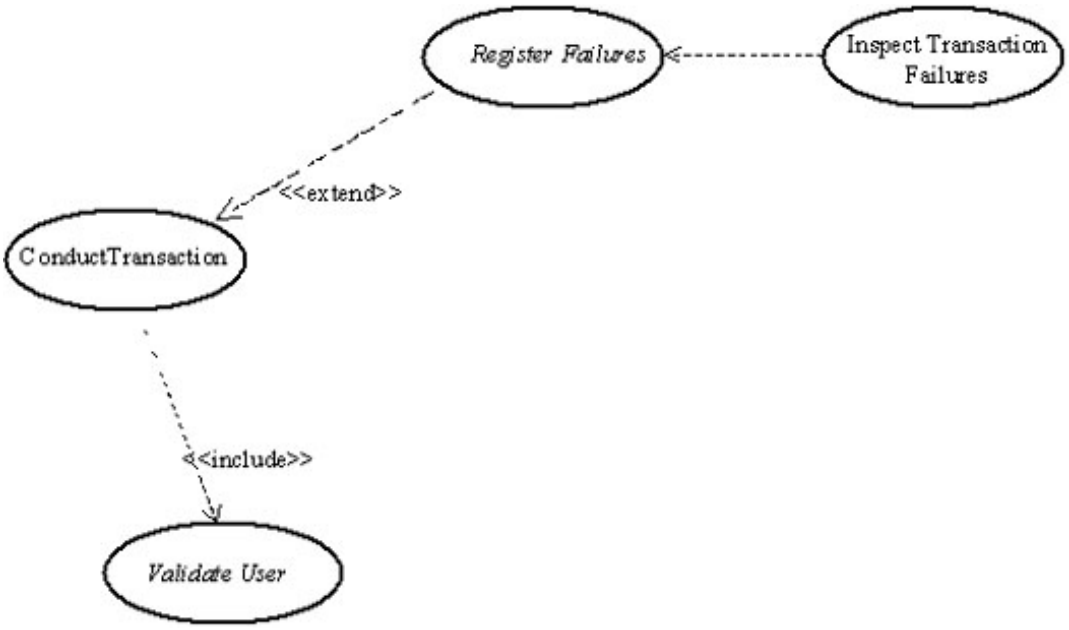

Figure 2: Part of a Use-Case Model with Two Concrete Use Cases (*Conduct Transaction and Inspect Transaction Failures*) and Two Abstract Use Cases, or Fragments (*Register Failures* and *Validate User*)

I have introduced a dependency between the concrete use case *Inspect Transaction Failures* and the extension use case *Register Failures*. Since this dependency is special and required only to attach an extension fragment to the use case that needs it, it may be a good idea to define a unique dependency stereotype (maybe <<need>>?) for it. But that is a separate issue.

The first class of use cases -- *concrete use cases* -- is fine; we don't need to do anything special about it. Concrete use cases can extend a base use case or be included in a base use case.

The second class of use cases -- the *use-case fragments*, or abstract use cases -- will be discussed below, along with suggested changes. However, first let's discuss a related subject: the relationship between extension and inclusion use cases.

Extension and Inclusion Use Cases Have a Lot in Common

Extension and inclusion use cases are *both related to a base use case*. During the "execution" of the base use case -- that is, when a use-case instance of the base runs -- it will (under certain conditions) interrupt the flow described in the base use case and instead follow the flow as specified by the extension or the inclusion use case. When the use-case instance has come to the end of the extension or the inclusion use case, it will return to the base use case and to the position in the flow described in the base use case, where it left off. This is true for both complete (concrete) and fragment (abstract) use cases.

The major difference between extension and inclusion use cases is the way the use case instance is instructed to interrupt the base use case and instead follow the extension or inclusion use case.

- *In the case of **inclusion**, the base flow itself explicitly instructs the use-case instance to obey the inclusion use case.*
- *In the case of **extension**, the base flow doesn't specify the interruption; instead, the extension use case specifies where in the base use case the use case instance shall make the interruption.*

The extension use case references an **extension point**, which specifies a unique location in the base use case. In OOSE and the Objectory Process, extension points belonged to the extending use case. In the work on UML 1.1, Jim Rumbaugh suggested that extension points should belong to the extended use case. The argument was for encapsulation: The extending use case should not see (it would be oblivious) the details of the base use case -- just the extension points. If you changed the extended use case, he said, only that use case would know the new location. I agreed with him.

Thus, extension and inclusion use cases are very similar; in fact, they could be considered *the inverse of each other.*

Proposal: Don't Call Fragments Use Cases

Neither extension use-case fragments nor inclusion use-case fragments are really use cases, and they should not be treated as use cases. Cleaning up this "defect" has long been overdue; we need to change the UML, by adding some minor notational elements.

Methodology users have the right to question any new notation. In the past, I struggled with "homegrown" methodologies that introduced notation for everything but didn't differentiate between syntax (notation) and semantics. Typically, their authors had no education in classic language design -- programming language or modeling language. Thanks to the work on SDL (Specification and Description Language) in 1981, and now UML, we have come a long way in developing more precise modeling languages.

Very simply: Classical language specifications (1) start from a *concrete syntactic construct* (a notational element in UML), which is (2) mapped into an *abstract syntactic construct* that, in turn, is (3) mapped onto a *semantic element*. The semantics specifies the meaning of the syntax. Most interesting syntactic constructs have a unique semantic correspondence. (The opposite is not necessarily true, since designed languages usually have many semantic elements [dynamic semantics] that don't have a syntactic correspondence.) I think it is standard language design practice to make every unique semantic element mappable from a unique syntactic construct. Natural languages are much more complex, but since we are creating the UML language ourselves, we don't need to complicate things.

Since fragments are NOT use cases, they should NOT be represented by the use-case syntax. That only makes it harder for analysts to distinguish among important elements. Fragments should be treated as they deserve to be treated -- as less important than real use cases. Note that I have no good proposal for what the new notational elements should look like. To

get started, however, I have chosen an icon that indicates a tiny element -- a dot. An icon that indicates a fragment would be more intuitive.

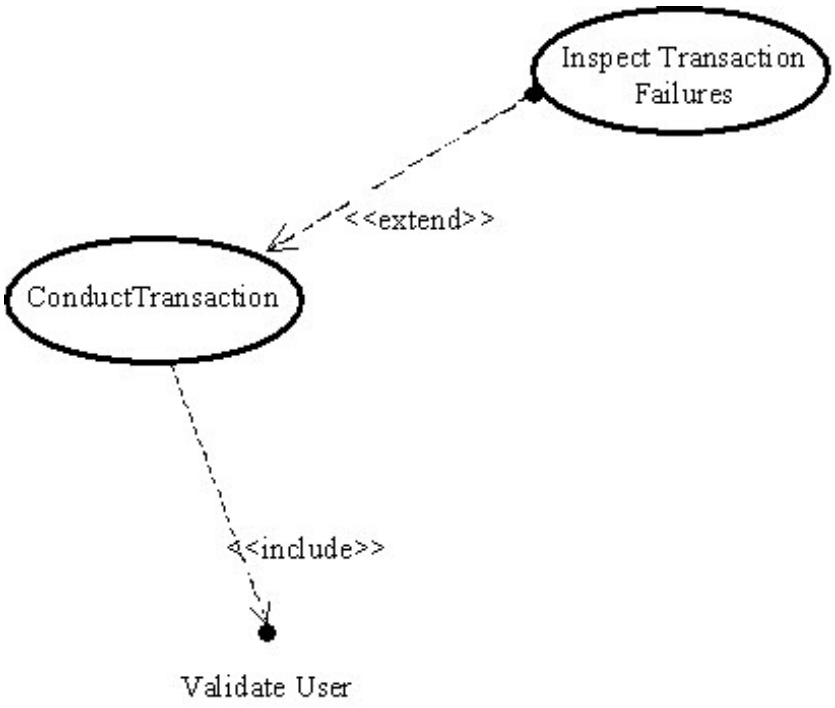

Figure 3: Treat Fragments Appropriately -- As Tiny Elements

In Figure 3, the *Register Failures* fragment in Figure 2 has collapsed to a dot, and its name has disappeared. The semantics of the dot are that when *Conduct Transaction* executes and reaches the extension point, something specified by the dot will happen: An extension will be executed. The *Inspect Transaction Failures* use case will use the extension to perform its responsibilities.

The dot represents an **extension fragment** -- the *Register Failures* fragment -- let's call it "E." E is not part of the real use case *Inspect Transaction Failures* (called UC2) that needs the extension. Since E is needed only by UC2, we don't need to give it a name. Instead, we attach it to UC2 by attaching the dot to the use-case symbol. However, it must be clear that the use case (UC2) that needs the extension E doesn't <<extend>> the other base use case *Conduct Transaction* (UC1) or the extension E. Thus, it would be completely wrong from a language point of view to have the use case UC2 <<extend>> UC1. Without the dot or something similar, we wouldn't be able to properly explain the desired semantics.

We can expand the dot (by "clicking the dot") to a new compartment of the use case. We could name the new compartment "Extensions," and use it to describe the extensions needed by the use case.

Our example has only one extension (*Register Failures*) that is needed by *Inspect Transaction Failures*, so we would describe it as a use case in the extension compartment of *Inspect Transaction Failures*. Since it is the only extension we wouldn't name it, but if there were multiple extensions, we might name the dots.

Sometimes an extension fragment is needed by several use cases (of type UC2). In these cases, the extension fragment must be named within the use-case model. The dot representing the extension fragment must be "free" from one particular use case, but related (via dependencies -- preferably using the new stereotype <<need>> or something similar) to all the use cases that need it. An extension fragment of this kind is a type of classifier with an extension compartment.

Figure 3 also has an **inclusion fragment**: the *Validate User* fragment. Inclusion fragments are not real use cases; they are reusable pieces of use-case descriptions or "text objects." Inclusion fragments are elements separate from the use cases that include them, so they are semantically similar to extension fragments needed by several real use cases.

There is an important difference between inclusion fragments and extension fragments:

- *An inclusion* fragment *will be "executed" by the use case instance that also "executes" the real use case (the one that includes it).*
- *An extension* fragment *will be "executed" by a use case other than the use-case instance that needs it.*

In the UML, a fragment should be a classifier with a notation that makes us think about tiny things -- but now I am going too deep for the purpose of this article. We also need a syntactic shortcut (syntactic sugar) for attaching a fragment to a concrete use case. We still need to use fragments in a pragmatic way.

The Day After Tomorrow: The Future of Use Cases

There have been many other ideas for improving use cases and their application over the years. I don't know them all, but below I'll describe some important ones and introduce two additional ideas of my own: making use cases code modules through aspect-oriented programming and making use cases run-time entities.

Add stereotypes for use cases. There are many ways to classify use cases -- as primary, secondary, and so forth; as business, software, or system, and so forth. Then, we can classify business use cases as supporting, managerial, or operational.[14] Using the UML stereotyping mechanism to classify these various types of use cases could help developers.

Clarify the relationship between patterns and use cases. Many design patterns are "templates" for reusable use cases. Such patterns are usually described using sequence diagrams or collaboration diagrams. A pattern is a solution to a general problem that can be applied in different contexts. There is thus an interesting relationship between a pattern and the generic, reusable use case that specifies the problem. Clarifying this relationship would be very helpful to developers.

Apply use cases within the domain of Human Computer Interaction

(HCI). HCI is a science. There is a way of designing a user experience by understanding the user community, its habits, and its metaphors. I was introduced to this technology by working with companies that were developing large commercial Web sites for huge user communities. Use cases could play an important role in integrating software development and HCI approaches.

Perform cost estimations based on use cases. In 1994, Magnus Christerson sponsored Gustaf Karner's master's thesis,[15] which resulted in a paper on project estimations based on use-case points (derived from function points). This is still an interesting paper, and with all the experience we have accumulated about use cases and project estimation, we should be able to modernize these ideas.

Start reusing use cases. This is a huge topic. Reuse of business software should start from understanding its use cases -- both those of the business, and those of the software to be used. This is the focus of the *Software Reuse* book that I published with Martin Griss and Patrik Jonsson back in 1997.[16] It is more relevant than ever today, when a company's IT support is built by integrating enterprise applications, whether these are legacy systems, packaged solutions, new applications, or Web services. I also discussed this further in my RUC 2002 talk, "Closing the Gap: Business Driven Enterprise Application Integration." (Available through Rational Developer Network: http://www.rdn.net; authorization required.)

Make use-case scenarios first-class citizens. It would be helpful to be able to identify and enumerate use-case scenarios, and to show dependencies between these scenarios. Recall that a scenario is *a use-case instance that we choose to model.* This is probably more of a process issue than a language issue, as there are many kinds of dependencies.

- **Iteration dependencies:** Each project iteration is driven by a number of use-case scenarios. Usually a use case is not completed within a single iteration, but is worked on over several iterations. I would like to be able to show how iterations are made up of scenarios, how a scenario grows over several iterations, and how several different scenarios over several iterations together make up a complete use case. You should be able to show, for instance, that you may have to develop less important scenarios first just to be able to develop more important scenarios later. As a concrete example, you may need to develop a use-case scenario that allows a telephone system operator to make a subscriber a valid user, *before* you develop any use-case scenarios that allow that subscriber to make telephone calls.
- **Test case dependencies.** There are other reasons for dependencies between scenarios. One is for testing. Integration test is built up test case by test case, in a manner very similar way to the way iterations are built up. We would be able to trace use-case scenarios to test cases. A good use-case scenario is a good test case. The relationship between a use-case approach and a test-first approach would become more streamlined.

Streamline use cases and aspects. One of the most exciting new movements today is Aspect-Oriented Programming (AOP). AOP was the buzzword of the year at OOPSLA 2002, and for very good reasons. Although I cannot discuss AOP in depth in this article, I do believe that AOP will bring about fantastic changes for use cases. The idea behind extension use cases -- to **add behavior to an existing system without changing it** -- is very similar to the idea of aspects. Use-case realizations are implemented as aspects, and extension use cases are realized as aspects. Extension points are semantically similar to "join points" in AOP.

When using RUP, within each iteration we specify use cases, design them, and test them. Between design and test we must disrupt the use-case flow in order to design, code, and test the components that together realize the use cases. Using AOP will simplify this: We'll go directly from use-case design to use-case programming, and then to use-case test. We will not completely get rid of component work, but dramatically reduce it. In effect, the use cases will be treated like modules that cross-cut the components. The work on these modules will be supported by our programming environment (an AOPL = an OOPL + aspects). AOP will allow us to seamlessly implement use cases.

Neither use-case driven development nor aspect-oriented programming is a silver bullet. They merely represent two best practices. However, I believe that integrating them will dramatically improve the way software is developed.

Make use cases run-time entities. The most exciting prospect for use cases is that they will also have counterparts in the run-time environment. Being able to identify executing use cases (use-case instances, in fact) in the operating system will help us with many important features. I discussed this in my 1985 thesis,[17] which had a semantic construct representing a use-case instance: A use-case instance was created by an external event, it lived during the whole interaction with the user (e.g., during a telephone call), and it tracked all the objects that participated in the use-case instance. With such a construct, we could change installed software much more incrementally -- one use case instance at a time. The software system could be restarted in much smaller steps, in most cases by restarting only the use-case instance that was broken. And, we could simplify the programming of use cases. With AOP we may achieve parts of this. Certainly, at least, we will achieve use-case oriented programming!

Final Words

Use cases have now been around for more than fifteen years. We can move them an important step forward by cleaning up a minor defect: separating use cases and fragments. This can be done tomorrow.

For the future, there are many other interesting ideas about how to improve use cases and expand their application. Many of these ideas represent marginal improvements, but two would be dramatic enhancements: making use cases code modules and making them executable run-time entities.

Until then, enjoy use cases as they are today!

Acknowledgments

I would like to thank Kurt Bittner, Gunnar Overgaard, and Paul Szymkowiak for their feedback on an early version of this article. Thanks to Catherine Southwood and Marlene Ellin for their editing.

Notes

1 These were customers of my company, Objectory AB, founded in 1987, which developed the Objectory Process over a period of eight years. When the company was acquired by Rational in 1995, the process grew, and it was renamed, becoming the Rational Objectory Process. Subsequently, the process grew even more, and again its name changed in 1998 to become the Rational Unified Process.

2 In this article, use cases are the focus of our discussion. Their importance in relation to other best practices (e.g., architecture first, iterative development) may therefore seem unbalanced.

3 Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

4 Again, these were customers of Objectory AB, primarily in Sweden. At the time we believed that, to really succeed with process, you needed to work closely with every customer throughout the entire lifecycle of a project. We had one or two consultants working full time with each project and were able to help the customer get started and address problems before they became serious. Customers included the Swedish Defense, Ericsson Mobile, Ericsson Radar Electronics, and ABB.

5 Kurt Bittner and Ian Spence. *Use Case Modeling*. Addison Wesley, 2003.

6 Jim Conallen, *Building Web Applications with UML*, (2nd Edition). Addison Wesley, 2002.

7 Peter Eeles, Kelli Houston, and Wojtek Kozaczynski, *Building J2EE Applications with the Rational Unified Process*. Addison Wesley, 2002.

8 Those of you interested in aspect-oriented programming will recognize that the intention of extensions is very similar to the intention of aspects in AOP.

9 For further information, look up "extends" in *Object-Oriented Software Engineering: A Use Case Driven Approach*.

10 Ivar Jacobson, "Language Support for Changeable Large Real Time Systems." OOPSLA86, ACM, Special Issue of Sigplan Notices, Vol. 21, No. 11, Nov. 1986.

11 We called them text items.

12 Gunnar Overgaard, Patrik Jonsson, Karin Palmkvist, and myself.

13 This example was provided by Kurt Bittner.

14 See Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology*. (Addison Wesley Longman, 1994). In this book we described business use cases in layers.

15Gustav Karner, "Use Case Points -- Resource Estimation for Objectory Projects." Objective Systems SF AB (copyright owned by Rational software), 1993.

16 Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Longman, 1997.

17 Ivar Jacobson. "Concepts for Modeling Large Real Time Systems." Dissertation,

Department of Computer Systems, The Royal Institute of Technology, Stockholm, September, 1985.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!